Article: Issues in choosing variables and methods for Program Performance in Java2

Author: Manivannan.P <manivannan.p (at) gmail.com>

Writing perfect programs needs careful choice of variables and methods. Java2 has numerous keywords to assign scope for members. Scope modifiers of members help in utilizing the rare compiler features of Java2 and boost up the performance. This article discusses various issues in choosing member variables, methods and their affairs with performance.

Importance of private scope:

Information hiding is a paramount of Object Oriented programming and this is achieved by *'private'* keyword in Java2. Some programmers do not specify any scope for members and leave blank declarations (or definition). However, it is poor practice, since it makes the members become accessible to other classes in the package.

Example:

```
class ABC
{
int id;   // 1
String password; //2
….
….
}
```

Class ABC has two members (id, password) with default scope public (i.e., here package scope.). So, it is possible for other classes (in the package) to gain access to these variables and manipulate them without any concern of ABC.

Hence, it is always better practice to declare variable scope as 'private' by default and make restricted access to them. Also access to these private members should be made possible only by the public methods of the same class.

Example:

```
class ABC
{
private int id;   // 1
private String password; // 2

public int getId() { return id ; } // 3
public String getPassword() { return password;} //4
….
….
}
```

Thus, the private members id, password become inaccessible for other classes and only accessible by respective 'public' methods.

Also, the compiler inlines the private members in some possible cases (not necessarily always). The inlined version of code is faster, since the variables are resolved in compile time. The inlining behavior for members is discussed later in this article.

public Vs performance:

A member with public scope is accessible at any point of the program code. Java2 disallows classes and interfaces to be declared with private or protected scope. Because, they will become inaccessible and cannot have instances (or references) for them.

As a common practice, sharable members can be declared as public.

Example:

```
class Example
{
public String message= “hello world” ; // the ‘message’ is accessible and mutable.
}
```

The problem with the public member is, it is mutable (by other class methods). Also, a public member has always trade-off with performance, since inlining behavior is not applicable. Also, variables can be resolved only during runtime. Hence, programmers should attempt to use public scope conservatively.

Role of ‘final’ keyword:

‘final’ variables:

The final keyword is used to declare (or define) members that are immutable (constant nature). Also, final indicates that the overriding of the member is banned.

The constants member variables are always intended to be declared as ‘final’, since they never change their values. Also, the compiler inlines the final members at compile time (not in all the cases), so that the program’s speed improves at runtime.

The following example illustrates the above in detail.

Ex:

```
class Blue
{
public static final int x =5 ;
}

public class Main
{
public static void main (String args [] )
{
```

```
System.out.println("x = " + Blue.x);
}
}
```

After the compilation of the classes 'Blue' and 'Main' classes, the Main classes look like this,

```
public class Main
{
public static void main (String args[])
{
System.out.println("x = " + 5);
}
}
```

i.e., the compiler automatically substitutes (inlines) the final variable 'x ', in the place of use. So, the variable x is resolved at compile time and hence, no need to lookup for 'x' value at runtime. This is a kind of slight optimization using final variable. This behavior will help when writing long final methods in classes.

'final' methods:

The 'final' keyword is used to declare methods to be free from future overriding. So, overriding final members is reported as compile time error.

In some cases final methods take roles in optimization. The compiler is pretty sure that a final method cannot be overridden in future, and hence there may not be any alternative version of it during runtime. So, the compiler tries to resolve the final method at compile time.

Example:

```
public final int getValue()
{
return 10;
}
```

Now, any call to the method getValue() is replaced by 10. For instance, the following statement,

```
int a = getValue()  + 5 ;
```

will look like,

```
int  a = 15;
```

after compilation. The reason is, the compiler wrapped the final method for inlining. But, this case is not applicable for all the final methods.

The compiler inlines some final methods which are pretty safe in runtime. One example is any final method with no local variables may be inlined at compile time. However, some smart JIT compilers inline even non-final methods at compile time. Also, they can un-inline if they find any dependency loading of classes. (This case is covered next)

Drawback of 'final':

Considering the same example,

```
class Blue
{
public static final int x =5 ;
}

public class Main
{
public static void main (String args [] )
{
System.out.println("x = " + Blue.x);
}
}
```

After the compilation of the classes 'Blue' and 'Main' classes, to run the Main class, it gives the output: x = 5.

Now, let the 'x' value in 'Blue' class be altered as follow,

```
class Blue
{
public static final int x = 8 ;
}
```

and leave the 'Main' class be unchanged. Lets compile only the 'Blue' class (Not 'Main') and then run the Main class, it gives the output as, x = 5.

The change made to 'x' value in 'Blue' class, does not take effect in 'Main' class. The reason is, the 'x' value used in 'Main' class is substituted (inlined) at compile time. So, the binary Main.class does not look back for 'x' value from 'Blue' class. So, the 'Main' class still prints the 'x' value as 5. Now, to compile the 'Main' class also, and run it again, gives the output: x = 8.

Hence, the 'final' keyword enforces the conditional compilation become mandatory. Otherwise, the pre-compiled binary will produce wrong output. Also, JLS (Java Language Specification, chapter-13) advices to use limited no of final variables; and the same effect of 'final' can be achieved by 'private static' combination.

Also, a general practice is to declare a variable as final, which never change its value (like, mathematical constants pi = 3.14).

'transient' (not) in persistence, serialization:

The keyword 'transient' is used rarely in programs. When class is stored in some persistence storage, the values of transient members of the class not stored. To give an example,

```
class T
{
int x;
int y;
transient String msg ;
}
```

When writing an object of class 'T' to a persistence disk, only the values of the members 'x' and 'y' are written and 'msg' is not written. However, the value of the variables during retrieval time is not specified by JLS. Also, no more explanation about transient is given by JLS.

From the serialization process (like persistence storage), one can interpret the following for transient members,

1. *'transient' members are re-initialized to their default type value during de-serialization (or retrieval time). Ex: 'int' type will be initialized to 0 , 'String' and 'char' are initialized to null and etc.*

2. *The transient variables do not take part in persistence state of object (also in serialization process). So, the transient variable's value is lost when de serializing.*

From above points, a programmer can intend to declare a variable as transient, when he does not want to store it. Variables used for storing intermediate values can be declared as transient. Also, variables that do not change the object behavior can be declared as 'transient'.

Usage of 'synchronized' keyword:

The 'synchronized' keyword is the first remedy for concurrency problem in Java2. When many threads try to access and manipulate a data object, the concurrency problems arise. There must be some 'lock' like mechanism to prevent the concurrency and exactly the same effect is achieved by 'synchronized' keyword.

```
public int  i=0;
public int changeValue()
{
++i;
….
```

```
…
return i;
}
```

When junk of threads makes random calls to the method changeValue(), the order of evaluation and the value of 'i' is pretty harder to predict.

So, 'synchronized' to keyword brings a lock like mechanism, called 'monitor' and locks, unlocks the method for each thread access. Locking and Unlocking the methods consume extra time.

Example:

```
public synchronized int changeValue()
{
++i;
….
…
return i;
}
```

The problem with 'synchronized' method is performance speed. Benchmark results proved that synchronized methods are very slow than normal methods (as per Jim Bell's benchmark result, in an ordinary case, a synchronized method is 10 times slower than a normal method).

An alternative to 'synchronized method' is 'synchronized block'. A synchronized block locks only a portion of a method (or program code in which the object lock needed), and makes the method become work like normal method.

Example:

```
public int changeValue()
{
synchronized()
{
++i;
}
….
…
return i;
}
```

However, a synchronized method has direct trade-off with speed. In this case, the programmer's choice is either speed or concurrency.

Usage of 'volatile' keyword:

The 'volatile' keyword is infrequently used in Java2. The volatile keyword can be used for mini-synchronization. The important thing is, the 'volatile' is not intended to give the same effect of 'synchronized', but the same nature.

When a variable is shared by many threads, there needs a sequence access to the variable. The following example depicts the case.

Example:

```
public int x = 5; //1
 ….
……
++x;            //10
```

the thread access to variable 'x' is undefined. When declaring the variable 'x' as volatile, there JVM internally gives exclusive access to its memory (but, no monitoring mechanism). When a thread access x's value, it can read the value only from memory in exclusive mode. So, a bit of concurrency is controlled in volatile variable access.

But, the volatile does not give the same effect of synchronized. To have better concurrency result, 'synchronized' is preferable than 'volatile'. For program speed, volatile is preferable.

Usage of 'static' keyword:

The 'static' keyword is admired a lot by the programmers. Often, they finger point 'static' scope for declaring constant natured variables. Since, the static variable is a common copy for all object space, it has direct advantage for memory and performance.

The common good source code of any programs, assign the static + private + final combination for constant variable's scope.

Example:

```
private final static int PI = 3.14 ;
private final static int SOUND = 333;
```

private, static, final combination brings out the maximum possibility for program performance.

Conclusion:

Declaring scope for variables is a trade-off at sometimes. Correct choice of scope modifiers makes the compile time optimization possible and gear up the performance in run time.

Reference

[1] The Java Language Specification, Second Edition. Authors: James Gosling, Bill Joy, Guy Steele, Gilad Bracha (**http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html**)

[2] Java Optimization, Jim Bell's benchmarking results. (**http://www.protomatter.com/nate/java-optimization**)

[3] Java Performance (**http://www.javaperformancetuning.com/**)

[4] Java™ Platform Performance Strategies and Tactics. Author: Steve Wilson, Jeff Kesselman. (**http://java.sun.com/docs/books/performance/1st_edition/html/JPTitle.fm.html**)

[5] The Java Virtual Machine Specification. (**http://java.sun.com/docs/books/vmspec/**)